Developing Multi Agent Systems using the Model Driven Architecture and Aspects

November 8, 2007

Toby Cox
tco17@student.canterbury.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

Supervisor: Dr. Richard Pascoe
richard.pascoe@canterbury.ac.nz

Abstract

Multi Agent Systems (MAS) comprise of a collection of autonomous and interacting agents that adapt to their environment. The agents within a MAS exhibit many of the same behaviours in the form of cross-cutting concerns. Aspects are a technology that can be used to represent cross-cutting concerns by weaving them through a system at specific points. In this paper a development process based on the Model Driven Architecture is proposed, that is used to perform a series of transformations from a platform independent to a MAS whose agency concerns are represented as aspects. The advantage of the proposed process is that a MAS containing many different types of agent can be easily modelled and transformed. Only the functional requirements of the MAS need be implemented after generation. The proposed process has been applied in small to medium MAS development scenarios with encouraging results obtained.

Contents

1 Introduction

Multi Agent Systems (MAS) are dynamic and loosely coupled communites of intelligent agents. In terms of software agents these agents ususally consist of expert systems; pieces of software that perform a specialised task (or set of tasks). The agent has a set of knowledge about the world, along with a set of rules which, based on input from those the knowledge, determine if and when certain actions should be performed. Software agents are usually loosely coupled since they communicate across a distributed medium, and hence must be flexible enough to cope with the non-deterministic nature of distributed communication. While they are loosely coupled, they must have common interfaces with each other to ensure that different variants of Agent reliably communicate with each other.

Agents within a MAS possess a homogeneous structure. For example, agents existing on a given platform are homogeneous in their use of that platform. Agents also possess commonalities in the general way that they behave. For example, most agents will initially register with a platform or similar in order to make themselves and their services available to other agents. The internal state of an agent is are affected and limited by the properties of the agent. These agency properties or agency concerns are the non-functional requirements of the agent.

Agency concerns are shared common across many of the agents within a MAS, yet are the core concern of none of these agents. These properties are therefore describes as cross-cutting concerns. The proposed process represents these cross-cutting concerns via the use of aspects. Aspects are a technology suited for representing cross-cutting concerns and reintroducing them to a system at specified "join points". Join points are specific points within the execution of code wherein an aspect can execute some behaviour.

The fact that agents are homogeneous in their usage of agency concerns, and that in order to provide agency concerns via aspects the agents within a MAS must have the required join points, is the motivation for the use of the Model Driven Architecture (MDA). The MDA is a process that details the transformation of a Platform Independent Model (PIM) into one or more platform specific implementations. The process proposed uses the MDA to develop the generic agents that make up a MAS with aspects are used to provide common functionality, as well as ensuring that each agent that requires a given agency concern has the necessary join points in order to allow an aspect to provide that concern.

1.1 Objectives and Goals

By modelling agency concerns as aspects with the MDA, reusable aspects can be used to separate the agency concerns of an agent from its core concerns. The goal of this research is to define a development process for MAS that allows a single PIM to be transformed into one or more PSMs, followed by a transformation from each PSM into the corresponding implementation. The agency concerns of the MAS shall be provided by weaving aspects with the generated MAS.
In order to keep this goal within the scope of this research, only one transformation to a single PSM shall be implemented.

Before the proposed process can be further elucidated upon it is necessary to describe some of the background concepts of the domain in greater detail. Section 2 contains further detail on agents, aspects and the Model Driven Architecture. This is followed by a description of previous work related to the

proposed process. The process is intended to be suitable for application across different agent platforms; and as such the description of the process in Section 4 is separate from any platform specific details. Section 5 contains descriptions of the tools used within the process. The application and evaluation of the process is carried out with two MAS scenarios and is described in section 6. Section 7 discusses some of the insights gained throughout the development of the process, and Section 8 summarises the success of the process and the direction of future research.

2 Background

2.1 Agents

Agents are usually expert systems that perform a specialised task based on their knowledge of the current state of their environment. The Oxford English dictionary [1] defines the word agent to mean *"a person or thing that acts or has the power to act."* A Multi Agent System (MAS) is a collection of agents which are generally involved in inter-agent communication, are aware of their shared environment and perhaps collaborate together.

Software agents are defined by the concept of agenthood[3] which is states that agents are *autonomous* entities that *interact* with their environment, and which *adapt* their state based on their interaction. By that definition, the following three properties are considered necessary for a software agent [4].

- **Autonomy** The property of autonomy means that the agent can exist independently. In pragmatic terms, the agent has its own control thread and accepts and rejects requests independently.

- **Interaction** Interaction means that the agent has some form of sensors and actuators[25] to provide input and output to their environment.

- **Adaptation** Adaptation means that the agent can adapt its state and behaviour based on cues from its environment.

In order for an agent to perform specific goals it may also require the functionality of agency concerns outside of these three primary properties. Such concerns might include learning, mobility or collaboration. These concerns are not central to an agents operation, but might be necessary in certain scenarios. Different types of agents will require different sets of concerns. The concerns of an agent are also related to the agents role within a MAS. A certain group of concerns is required for an agent to able able to take on a certain role. For instance; an agent wanting to assume the role of a collaboration participant might need to have the three core concerns as well as the collaboration concern and perhaps the learning concern. Conversely, roles can be used to describe a set of concerns. If an agent is known to have assumed the collaboration participant role then it can be assumed that the agent contains the five concerns associated with that role.

Agency concerns are a platform independent term as long as they are defined as behaviours which are shared between a community of agents of any kind, not necessarily just software agents. A community of agents could be imagined that consisted of a group of cleaners cleaning a building; an agency concern could represent the need for each agent to communicate with the others in order to ensure that they do not clean the same area twice. The description of this necessary behaviour without defining how the behaviour is actually implemented is where the term agency concerns is used.

A key concept of an agency concern is that the behaviour is shared across multiple agents; this is known as a cross-cutting concern. Abstracting the agency concerns using conventional techniques such as object oriented components is difficult. Object Oriented Programming (OOP) and design patterns such as the Strategy pattern [9] can be used to abstract the behaviour; but agents then develop a type of object schizophrenia [10] in which the state of an agent and its behaviour become separated into multiple objects, where they are expected to be one. OOP is not suited capturing cross-cutting concerns [14], and as such a different paradigm for representing agency concerns is required.

2.2 Aspects

Aspects[19] are a technology which allow cross-cutting concerns to be captured and weaved across code. They differ from design patterns in that they provide the localisation and of a concern based on a requirements or platform constraint. Whereas design patterns provide localisation of a concern in terms of a technical problem [16]. Aspects capture cross-cutting concerns and provide them via a weaving process across the base code. The base code that the aspects are weaved with is generally oblivious to the fact that it is being targeted by an aspect(s). The weaving process applies the functionality at points in the execution of the code known as join points. These points are targeted by a construct called a pointcut which is essentially a form of pattern matching. The aspect associates advice with the join point targeted by the pointcut. The advice defines an operation that is to occur at the execution of code at the join point. In order to combine the cross-cutting concern that the aspect represents and the core functionality of the system, the aspect is weaved with the program. This weaving process can occur at run-time or compile-time and results in a program that executes with the union of the system's functionality and the aspect's functionality.

As Yu et al [17] state "Software using Aspects is easier to develop, understand and maintain". This is primarily due to the fact that aspects simplify an implementation by separating the cross-cutting concerns from the functional requirements code. In the case of MAS development this means the developer need only deal with the implementation directly related to the functional requirements of the agents.

Aspects can represent both the functional and non-functional requirements of a system [19, 5]. Non-functional requirements are global properties of a system that both dictate and limit that capabilities of a system, whereas functional requirements specify actual behaviour of the system.

The process proposed uses aspects to capture agency concerns. The aspects are then weaved with the agents in the MAS that require the agency concerns. Different implementations of the agency concern aspects are weaved with the MAS in order to provide differing behaviours to the agents. This ensures that from the developer's point of view the implementation of the agents is focused primarily on the functional requirements.

2.3 The Model Driven Architecture

The proposed process represents agency concerns via aspects, but in order for the aspects to be able to apply the agency concerns, the correct join points must exist within the agents. The code that comprises a join point is also known as glue-code [11]. It is so called because it is the point at which additional functionality is affixed to the system. These points are required for any type of additional functionality to be added to a system [14]. The MDA process is used to reliably provide this glue-code within a generated MAS. The Model Driven Architecture (MDA) can ensure this by ensuring that points within the implementation of the MAS that the aspects target are consistently available.

The MDA is a set of transformations defined by the OMG[1]. The first step in the transformation process is the definition of a Platform Independent Model (PIM). This model defines the system in completely abstract terms; within the PIM there are no platform specific implemention details whatsoever. This model

[1]The OMG is a consortium focused on modeling (programs, systems and business processes) and model-based standards.

describes the structure of the system and the interactions between components. All models in the MDA process are defined using the Unified Modelling Language (UML).

The next stage in the process involves the transformation of the PIM to the Platform Specific Model(s). The Platform Specific Model is a representation of the same business logic as the PIM; but with some platform specific information contained. This platform specific information might take the form of platform specific types, messages or notation. The platform specific model should have enough information for a developer to take the model and implement the system on the related platform with no further information.

The transformation from PIM to PSM can be fully automated. The MDA specification does not specify the technical details of the actual transformation; this is implemented by specific MDA tools. Most MDA transformation tools transform the XML Meta Interchange (XMI) data that represents the PIM into the XMI of the PSM. Once the PSM has been generated, modelling tools can be used to add further platform specific detail.

The last stage in the process is the transformation from the PSM to the implementation. Again, this transformation can be fully or partially automated using MDA compliant source code generation tools. As it is difficult for the PSM to completely represent the implementation, the generated implementation is usually subject to some manual development.

We have outlined three distinct layers here (PIM, PSM and implementation), but there is a fourth and more abstract level of model that is not incorporated in the process. This model is called the Computation Independent Model (CIM). The CIM is essentially an ontology that represents the domain in a vocabulary familiar to practitioners of the domain. This model does not detail the architecture necessary for the functionality that the requirements dictate. The CIM has not been explicitly included in the process as the PIM has similar semantics to the CIM described in the literature [15]. The PIM can be considered to be akin in some ways to a CIM as it contains no mention of platform specific concepts such as aspects and agents. This is discussed further in Section 4.1.

3 Previous work

Garcia et al [11] talk of using aspects to represent agency concerns. They define agency concerns as including, but not being limited to, things such as Interaction, Adaptation, Autonomy, Learning and Mobility. They propose a unified model in which agency concerns can be introduced to the object model using aspects. They are engaged in a more abstract discussion and outline as future work is the need for code generation tools with relation to aspects and agency concerns. The proposed process includes this code generation via the MDA process.

Garcia et al also use the terms "quantification" and "obliviousness" These are described as being fundamental properties of aspectual modules as coined by Filman [7]. Quantification describes the statements scattered throughout a programming system that perform essentially the same task. This is essentially a synonym for a cross-cutting concern. Obliviousness is the concept that the software that is the target of the aspect is unaware that it is the target of an aspect and that by examining the base code you cannot conclude that an aspect will execute in a certain place. These two concepts provide a basis for determining the suitability of an agency concern for representation via aspect. Garcia et al informally use stereotype UML annotations to represent aspects, but do not formally define a method of defining agency concern aspects in the UML.

Ho et al [13] however do make an effort to define the representation of aspects in UML. They describe UMLAUT, which is a framework/tool for manipulating UML models. Using UML stereotypes aspects can be defined in the UML, after which the UMLAUT tool performs an automated weaving process. Following this another weave can be performed to derive the implementation model. This results in a UML model that represents the aspects as design patterns. Ho et al also talk of the differing dimensions of modelling aspects. They describe four dimensions: functional, static, dynamic and physical. The proposed approach focuses only on the static dimension of modelling; the dimension related to UML class diagrams.

Gonzales et al [6] also talk about dimensions, but in a different context. The concept of dimensions they describe is related to the way in which cross-cutting concerns are related and the dependencies between them. When decomposing a system's concerns into components or aspects, these dependencies must be recognised and dealt with. Gonzales et al [6] provide an overview of multi-dimensional decomposition, that is, decomposing the multiple concerns of a system optimally. This concept is important when identifying agency concern aspects and ensuring that the aspects do not conflict and negatively affect the performance of the MAS. Rashid et al [24] also describe an approach for modularising concerns and determining conflicts, but they do so at the requirements level and focus more on creating an XML-based description language.

The concept of transformations between models as described in Ho et al and the combination of this transformation process with aspects has been touched on in several previous publications [8, 21, 6, 20]. The proposed process is similar to Ho et al [13] but employs the MDA process.

Fuentes et al [8] talk of using the MDA to create an aspect based system. They define a set of MDA transformations to work through a set of PIM models. The first model is the Computational Model (C-M), this model is essentially a set of objects interacting via interfaces and messages, these objects may have

constraints defined that essentially are cross-cutting concerns. The second is the Component and Aspect Model (CAM), this model defines basic entities of the system from an architectural point of view. Components and aspects are the basic building blocks of this model. The Dynamic Aspect-Oriented Platform (DAOP) is a model that represents a system as a set of components and aspects which are treated as first class reusable entities. The middleware platform is the final model, and is technology dependent. In the terms of the MDA, this model is the PSM. The proposed approach is similar in that both the technologies of the MDA and aspects are used. However since the focus is on using these technologies to develop a MAS, the aspects developed can be more specific to the domain and the interface between the aspects and the base-code can be less general.

In order to model aspects using UML, a suitable UML profile must be used. Omar et al [2] propose an aspect UML profile using stereotypes, tagged values and constraints to allow UML to model the concepts of aspects. Their profile augments UML while leaving its original semantics. The profile models aspects as stereotyped classes and can classify the relation between aspects and core classes into synchronous and asynchronous. Synchronous aspects can control the internal behaviour of classes, while asynchronous aspects usually perform their own task and have no impact on the the core classes. These twin concepts have an effect on the suitability of an aspectual representation of an agency concern.

Robbes et al [25] describe a similar process to part of the process proposed here. They use aspects to capture concerns within a MAS; however they represent concerns at a level of abstraction they call a role. A role encompasses certain behaviour and and is weaved with an agent so that the agent may take on that role in the MAS. The process proposed uses aspects to represent behaviour at a more fine grained level; roles as defined by Robbes et al are implicitly defined by the behaviours that an agent exhibits. The OPAL agent platform also contains the concept of Roles, but it uses them as essentially a classification of agent types. Since the concept of a role is already incorporated in OPAL (albeit with slightly different semantics) and the proposed process encapsulates agency concerns using aspects; the concept of Role is not as relevant. However it is conceivable that different MAS platforms that do not incorporate the concept of Roles could benefit from having Roles represented using aspects in this way.

4 The Proposed Process

The proposed process involves three phases taken from the MDA and a fourth that is the union of the generated MAS and the agency concern aspects. They are:

1. **Platform Independent Model (PIM)**
2. **Platform Specific Model (PSM)**
3. **Generated Implementation**
4. **Completed MAS**

Figure 4.1 outlines the entire process. Individual steps are elaborated upon in the following Sections.

4.1 Platform Independent Model

The first step in the process is to create a platform independent model of the MAS. This model can encompass both the static structure of the MAS as well as the dynamic components such as conversations etc[1]. This PIM models the entities that will become agents in the final implementation, but it does not contain any notion of agents. This is to ensure that the PIM shows no bias to any particular platform, to the point that the PIM applies just as much to an agent paradigm as it would to any other paradigm. While it could be argued that agent is not a platform specific term, in that the PIM need not be limited to describing software agents; the decision was made decided to refrain altogether from using the term in the PIM in order to limit potential confusion. Accordingly only "behaviours" are defined within the PIM, rather than agency concerns.

4.1.1 UML profile

In order to fully model the PIM, an Agent UML profile has been defined that augments the modelling tool. This profile captures the concept of agency concerns without actually using the term agency. The term behaviour fragments was chosen, based on previous work in Whitestein technologies' Agent Modelling Language (AML) profile [2]. The profile defines these behaviour fragments as stereotyped classes. Figure 4.2 shows the simple AML profile developed. Using these constructs the types of agent within the MAS can be modelled according to the behaviour required.

Defining behaviour fragments is a simple way of modelling Agency concerns within the PIM, but it is also necessary to specify a way of associating entities with those behaviour fragments. Several techniques for association were examined.

The first is based on the AML examples provided by Whitestien technologies. They associate behaviour fragments with entities using stereotyped attributes. This is a simple and obvious method, but pollutes the model with superfluous attributes.

[1] Our process currently transforms only the static elements of the model. This is discussed further in Section 7.

[2] Whitestien Technologies, AML Specification version 0.9 - http://www.whitestein.com/library/company-and-product-resources

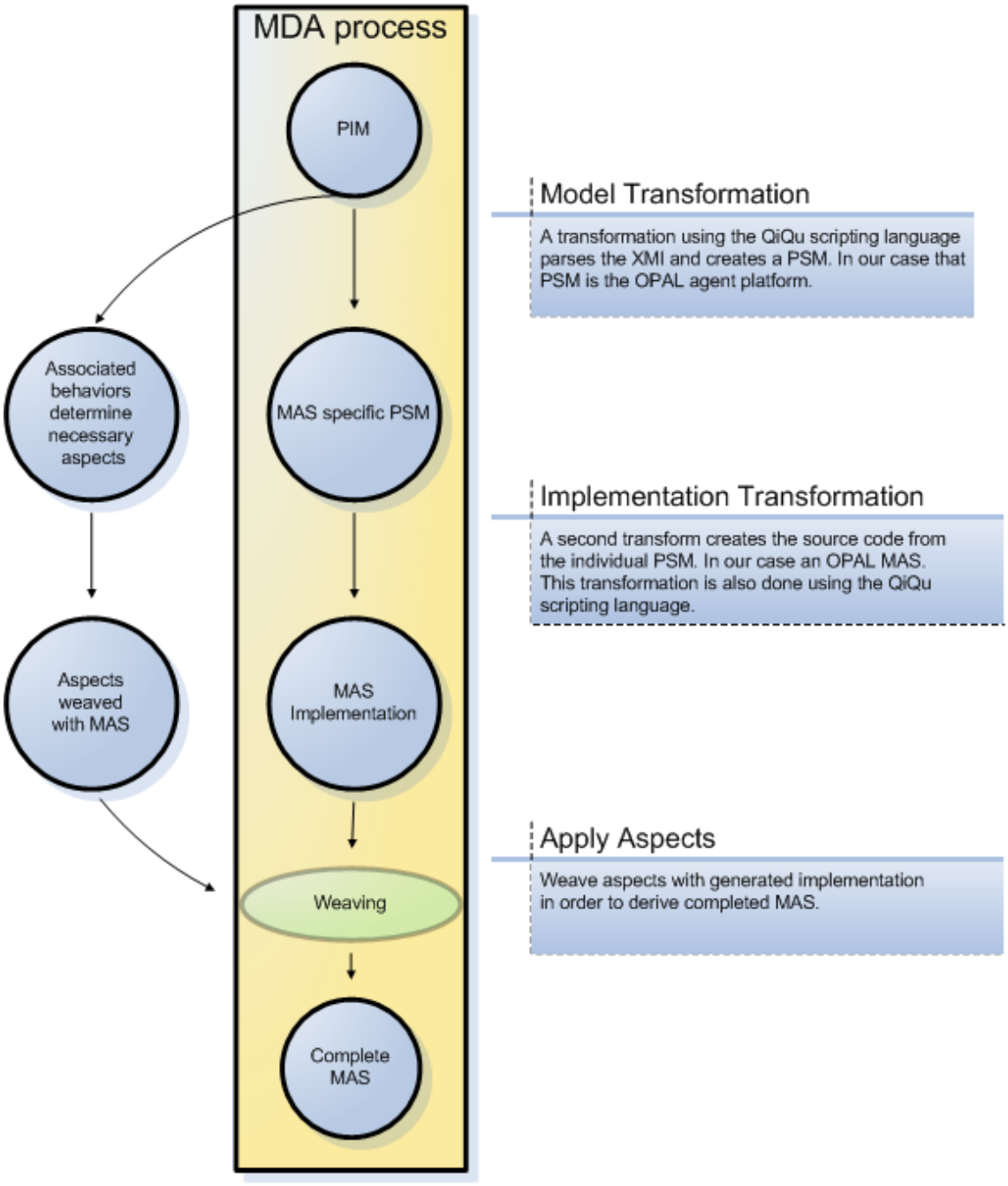

Figure 4.1: An overview of the entire proposed process.

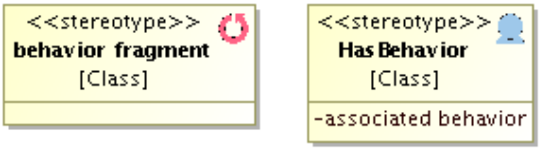

Figure 4.2: The AML profile developed to model the PSM.

Another option is to add a tagged value definition to the entity in the AML profile. Any behaviour fragments used by a given entity can be specified in this tagged value. This option is clean and does not interfere with the model, but requires the modification of the AML profile in order to specify the tagged values.

The third option is to simply specify a UML association between an entity and the behaviour fragment it uses. This solution, while not as elegant as specifying tagged values, is easier to implement and more visually interpretable. It does however have the distinct disadvantage of adding visual load to the diagram. In a large model with many entities using the same behaviour fragments, the diagram would quickly become cluttered.

The latter two techniques were both implemented, with the tagged value approach being chosen. This was partly due to the fact that the modelling tool used facilitated the modification of the UML profile well; specifically the definition of tagged values. If constraints were placed on modelling tools then the method involving UML associations would be easier to implement, if less elegant.

Using the tagged values technique, entities in the model that semantically represent agents are given the stereotype *HasBehavior* which has an *associated behaviour* tagged value. This tagged value is populated with the names of the behaviours which the stereotyped entity requires.

4.2 Platform Specific Model

The transformation to the PSM takes the XMI representation of the PIM and creates an XMI representation of a MAS specific PSM. This model may describe things such as the generalisation of agents or agent communication that is not part of an agency concern.

The two primary tasks in the translation are transforming the general Agent related constructs into platform specific constructs, and creating suitable interfaces for aspects in order to provide the required behaviour fragments. A transformation was employed to select all entities in the PSM stereotyped with $<< HasBehavior >>$. Since these entities have behaviour and the PSM being generated is an agent PSM, then these classes should be represented as agents in the PSM. The transformation ensures that the agents have the correct requirements for agency on the given platform. For example, an agent might be required to extend a more general agent class in order to be considered an agent a given platform.

The second task involves identifying the Behaviour fragments associated with each Agent and ensuring each Agent has the required interface for the aspect which implements that behaviour fragment.

4.2.1 Defining Aspect Interfaces

It is important that the PSM define clear and flexible interfaces for the agency concerns, and it is desirable for the agency concerns to be reusable between agents within the same MAS and also similarly defined Agents in other MAS. If Agency concerns are not reusable the need to represent them separately is diminished.

In order for aspects to be able to be able to cleanly represent the required behaviour of the cross-cutting concern, interfaces must be defined through which the aspect can provide all required functionality and which will be the target of the pointcut. It should be noted that these interfaces are not necessarily implemented using Object Oriented definition of interface. They are simply an interface between the cross-cutting concern and the core functionality of the MAS; this is the concept of glue-code discussed in Section 2.3. The proposed process provides these interfaces via method stubs that are targeted by the agency concern aspects. In the PSM it is necessary to represent these methods somehow in order to ensure they are present in the implementation.

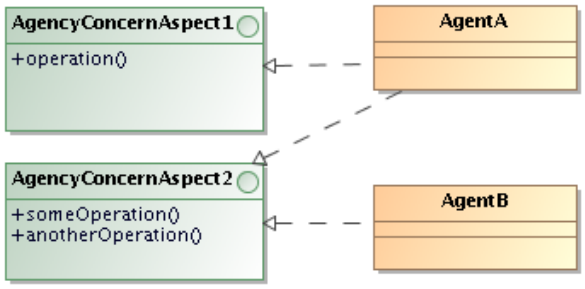

Figure 4.3: Agents are related to their agency concerns in the PSM via the use of interfaces which ensure that agency concern aspects can target the agents.

Initially the use of annotated methods could provide a clear and simple way of representing the method stubs. The annotation on a method would indicate that the corresponding agency concern aspect should be weaved with that method. Since aspects can target methods based on annotations, it would seem that representing the stubs as annotated methods in the PSM would be a simple solution.

However further investigation found no formal method for representing annotations in UML. This meant that the annotations would have to represented in some other way in the PSM and then translated to annotations in the implementation, which was not desirable.

Instead actual Java interfaces were employed for the task. Agents in the PSM realise the corresponding interface for the aspect that proves the behaviour they require. The interface forces the agent to implement a basic set of method stubs. These stubs are generated by the MDA process, without any implementation. The aspects that represent the agency concerns use the methods enforced by the interface as join points for the interface. Figure 4.3 illustrates how this is represented in the PSM. In the PSM described in figure 4.3 *AgentA* requires the agency concern provided by both *AgencyConcernAspect*1 and *AgencyConcernAspect*2; whereas *AgentB* only requires *AgencyConcernAspect*1.

The decision to require agents to implement interfaces in order to make use of the agency concerns aspects was made for two reasons.

The first reason was the ease of code generation. The PSM is generally transformed directly into source code in an automated manner. As such, constructs in the PSM should be directly relatable to artifacts in the source code. Since the aim is to provide basic skeleton methods for aspects to target in the implementation, interfaces provide the most natural way of ensuring that those methods exist in the implementation. Any Java/C++/C# style source code generation tool that takes XMI as an input should be able to provide an implementation that can be targeted by aspects of the targeted platform. If a solution such as marking the agents with a stereotype or similar and having the transformation process provide the necessary artifacts for aspect join points has been chosen; then the generation process would need to be specialised.

The second reason for employing interfaces was familiarity. Object Oriented designers are familiar with the concept of interfaces and they are a standard part of the UML. An interface generally implies that some behaviour (a method) is compulsory for that class if the class implements that interface. Typically interfaces are implemented so that clients of a classes can rely on the static nature of that interface; and will not be affected by changes in the underlying implementation. In the proposed process the semantics of an interface are somewhat different, as the interfaces are no longer being implemented for the sake of external clients. Since agents are autonomous entities, external clients do not generally use method invocation on public methods of the agent. The implemented interfaces are primarily for internal use for the agent itself; as well as being the join point for the required aspect. For example: an agent might require the communication behaviour fragment, so the implementation transformation would ensure that the agent

realises the communication interface, which defines the *sendMessage* and *processRequest* methods. These are generated as method stubs to ensure the interface is conformed to. The *Communication* aspect uses these two methods as join points in order to provide advice which controls the sending and receiving of messages. The *Communication* aspect is defined in more detail in Section 6.2.

4.2.2 Manual Modification of the PSM

After transforming the PIM to derive a PSM, it may be necessary to add or extend the resultant model in order to flesh out the functional requirements. This additional modelling is related to the gap between how much of a complete system an automated process can produce. This gap can exist at either the PSM or implementation level. It can be narrowed at the PSM level by deeper modelling of the behaviour of classes. A balance must be found between the amount of effort expended in creating a detailed model and transformation to create the desired source code, and the value gained from having this source code being generated automatically. It is difficult to fully close this gap due to the complexity of fully automating a transformation from a complex model to a potentially more complex implementation. To narrow this gap in the process, the transformed XMI is imported back into a modelling tool. Any further concepts that could not be defined at a platform independent level and that can be modelled at the platform specific level are then added. Generally speaking, the aim at this step is to model the agents of the MAS such that they can be as completely as possibly implemented on the chosen platform. Looking forward one step in the process, to the automated transformation of the PSM into source code; reveals that the PSM must be modeled in such a way that the automation process can be successful and complete. This means that the PSM must be created using UML that is suitable for input into existing source code generation tools.

4.3 Generated Implementation

Once the PSM is suitably defined the next step is transform it into source code. This process can be highly automated and if the PSM was well defined then existing source code generation tools can be used. This step of the process results in a set of agents as originally described in the PIM. For the most part the agents generated are empty except for the method stubs necessary for the agency concerns aspects. Any other generated implementation will have come from manual additions to the PSM. The following is a simplified version of the implementation of *AgentB* generated from the PSM in figure 4.3:

```
public class AgentB extends OpalAgent implements AgencyConcernAspect2{

    public AgentB(String displayName){

    }

    public String someOperation(){
        return null;
    }

    public String anotherOperation(){
        return null;
    }
}
```

The methods *someOperation* and *anotherOperation* are realised from the *AgencyConcernAspect*2 interface and are the join points for aspect related to the *AgencyConcernAspect*2 interface.

4.4 Completed MAS

The final step involves taking the generated aspects and applying them to the generated MAS. The aspects weave with the MAS and provide the agency concerns of the MAS at the points provided by the aspect interfaces. Agents use the agency concerns via the methods realised from the interfaces. The agents themselves are oblivious to the fact that the functionality of the methods they call is being provided by aspects.

5 Tools to Support the Process

5.1 Tools

In the previous section the proposed process was described without mention of the specific tools necessary at each stage. The process requires tools for both the transformations between stages of the MDA process, and modelling tools for creating and editing the required models.

5.1.1 Modelling Tools

The proposed process requires a modelling tool in order to create the PIM and to augment and modify the PSM. The modelling tool must also be able to export, and ideally import, XML Metadata Interchange (XMI) files. XMI is an XML based representation of UML models. It is a core part of the MDA process as it is the format in which models and distributed and manipulated.

StarUML

Initially StarUML[1] seemed like a suitable candidate for use as the primary modelling tool. StarUML has the benefit of being open source as well as having an AML (Agent Modeling Language) Profile, which provides notation for modelling MAS. It provides a complete suite of UML widgets as well as allowing UML profiles to be added in order to extend its modelling capabilities. StarUML provides MDA support through the automatic generation of source code from the modelled PSM.

The StarUML AML profile provides the notation necessary for modelling MAS as well as providing icons to clarify the added Agent semantics. The profile uses a set of Stereotypes to define MAS components. For example; a class is declared as being an agent by stereotyping it as $<< agent >>$. The behaviour of Agents is modelled using a behaviour fragment. This construct allows for the representation of Agent behaviour. Essentially these are used to model Agency concerns. This is highly useful the goal is to model Agency concerns as aspects, and behaviour fragments provide a construct for modelling these at an abstract PIM level.

Initial models developed using StarUML quickly revealed the limitations of the software. The exporting of XMI is critical in the MDA process, as it is the XMI that is the subject of the PIM to PSM transformation. StarUML crashes unpredictably upon encountering certain UML constructs. StarUML also exports the older XMI 1.1 for UML 1.3, rather than the newer XMI 2.x for UML 2.x. It quickly became apparent that this would be an issue, as most source code generation tools take the newer version as input, and there is a considerable difference in the two formats. As a result, a different modelling tool was chosen.

MagicDraw

MagicDraw[2] is a commercial UML modelling tool that provides a complete set of UML notation, as well as third-party UML profiles. It does not however have an Agent UML profile available. This is not a huge problem as modelling a small MAS requires only a subset of a complete Agent UML profile. The necessary

[1] StarUML - http://staruml.sourceforge.net/en/
[2] MagicDraw - http://www.magicdraw.com/

constructs can be created and combined to form a smaller UML profile. As such, MagicDraw was used and a small Agent UML profile containing the necessary constructs was implemented.

In creating a UML profile for MagicDraw some of the constructs encountered in the AML profile for StarUML were drawn upon. Initially an *Agent* stereotype was used for modelling agents in the PIM, but after deciding to keep the PIM as platform agnostic as possible a *HasBehavior* stereotype was employed instead. When applied to a class this stereotype implicitly signifies an agent, as in terms of entities in a MAS only agents have behaviour. The other primary construct was $Behaviour fragments$, which were modelled as stereotyped classes.

5.1.2 Transformation Tools

QiQu

QiQu [3] is a powerful scripting language designed with the MDA in mind. It provides flexible manipulation of XMI using a combination of Xpath queries and its own syntax. QiQu is especially useful in performing the PIM to PSM transformation in the MDA process; but it can also be used to transform the PSM into an implementation. The PIM to PSM transformation process is especially easy using QiQu because it simply involves modification of the XMI. QiQu is especially adept at this task. Using QiQu for the PSM to implementation transformation involves recognising constructs in the PSM XMI and outputting the relevant code. This can either be done by outputting each agent class manually; or by the use of Velocity templates, to streamline the generation process by providing class templates which a QiQu script can provide variable to.

AndroMDA

AndroMDA[4] is a framework for generation of implementations from models. It that is a very popular framework, especially in association with MagicDraw. AndroMDA supports code generation via a "cartridge" framework. Cartridges can be swapped in and out of AndroMDA allowing it to generate source for any target platform. The cartridges can be easily extended or developed from scratch. AndroMDA provides a UML profile which can be used by MagicDraw to model PSMs.

While AndroMDA initially seemed like a good candidate for implementation generation, actually implementing it in the process proved difficult. AndroMDA proved more oriented towards the developments of web applications and installation and configuration was troublesome. Furthermore AndroMDA projects are intially generated with a blank model; the modelling of the system is done in this file. This did not tie in well with the process which requires that an XMI file be fed to the generation transformation. Also limitations in the modelling tools used limited the ability to bypass this difficulty.

Applying the Modelling Tool to the Process

Because of the limitations of AndroMDA, QiQu was used to perform both the PIM to PSM translation as well as the PSM to implementation transformation. While not as elegant or flexible as a generic source code generate tool, QiQu was sufficient for proof-of-concept purposes. It is likely that AndroMDA could be included into the process as a generation tool, but more time would need to be spent determining how to it should be configured.

Figure 5.1 details how the modelling tools and the generation tools were applied through the process.

5.2 Aspect Framework

Aspect are not native to the Java programming language, and so a suitable aspect framework is necessary. AspectJ [18] was chosen due to previous experience, and because it is a popular aspect implementation

[3] QiQu MDA scripting language - http://www.aloba.ch/qiqu/
[4] AndroMDA - http://www.andromda.org/

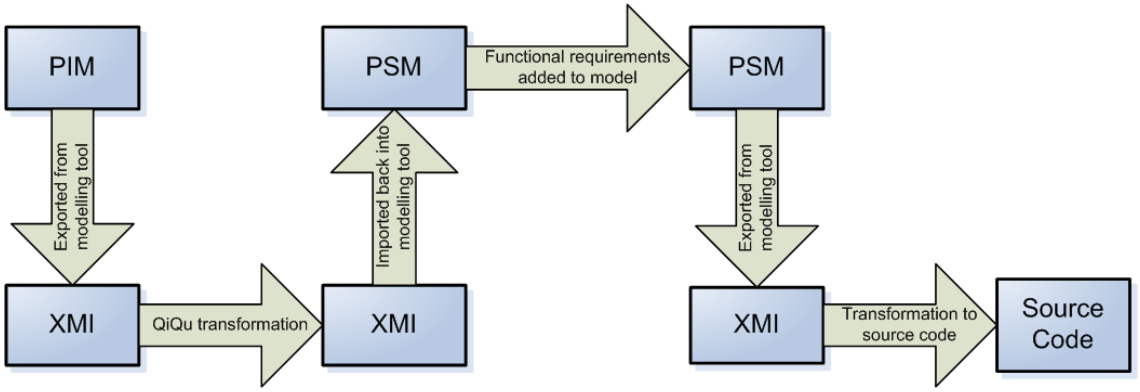

Figure 5.1: The proposed process from a tool viewpoint.

which provides complete support for Aspect Oriented Programming (AOP). Other aspect implementations exist [22, 12], but AspectJ is a mature implementation and has integrated well with the proposed process.

An AspectJ aspect is primarily composed of two things: pointcuts and advice. A single aspect may contain more than one of each. The point cut of an aspect is a group of points known as join points. Join points are well defined descriptions of a point in the execution of the program. AspectJ join points can target things such as methods or variable declarations. Pointcuts essentially use pattern matching in order to target specific join points. A pointcut targetting a method can match on method name, parameters, return type as well as annotations. The matching can include wildcards for any of these parameters, as well as acknowledging polymorphism. A basic example of a pointcut might be:

```
pointcut fooBarPointcut(int i):
 call(@SomeAnnotation public String Foo.bar(int)) && args(i);
 }
```

This pointcut matches any calls to methods named *bar* who are members of the class *Foo*, who return a String and take a single *int* as a parameter. The arguments of the function are exposed to any related advice via the $\&\&args(i)$; section of the pointcut. For this pointcut to correctly match, the targeted method must be annotated with the *SomeAnnotation* annotation.

The second part of the aspect is the advice. The bulk of the advice is method-like code which captures the functionality of the cross-cutting concern. This code is wrapped by the advice and related to one of the aspect's pointcuts. There are three basic types of advice: *before*, *after* and *around*. The type of advice defines the point at which the advice's code is executed; naturally *Before* advice executes before and *after* advice executes after the pointcut. *Around* advice has the special condition of selectively preempting the the normal action at the join point. *After* advice also has the special conditions of *after returning* and *after throwing* which correspond to the two ways in which a method can return.

AspectJ has one other useful feature, which is inter-type declarations. AspectJ aspects can declare members (fields, methods, and constructors) that are owned by other types. These are called inter-type members. Aspects can also declare that other types implement new interfaces or extend a new class.

6 Application and Evaluation

Two MAS were developed in order to evaluate the effectiveness of the proposed process. Each MAS was based around a different scenario and each implementation was obtained entirely using the proposed process. The goal of the evaluation was to determine if the two different MAS could be developed using the same proposed process, and using the same aspects to provide agency concerns. The ability to transparently swap agency concern aspects was also evaluated by implementing a single agency concern aspect in two different ways. Both MAS were targeted at the OPAL agent platform using the Java programming language, and both MAS used the same set of agency concerns. The OPAL agent platform is described in Section 6.1, followed by a description of the agency concerns that were implemented as aspects. The rest of this chapter is dedicated to the description of the two MAS scenarios implemented using the proposed process.

6.1 The OPAL agent platform

The OPAL agent platform [23] is an architecture used to represent software agents. Agents can be represented at multiple levels of abstraction; from micro-agents at the lowest level, to more sophisticated agents consisting of a set of micro-agents. The OPAL agent platform was chosen due to previous experience, and the fact that agents can be developed quite quickly and simply. While the actual structure of the Opal architecture is not important within the scope of this research, it is helpful to have a basic understanding.
An OPAL MAS has at least one Platform agent. When an agent is created it is registered with its local Platform. The Platform consists of micro-agents that provide services such as directory services and messaging. Agents can also register with remote platforms in order to notify those platforms of their existence.

6.2 Implemented Agency Concerns

A large part of this project involves identifying cross-cutting concerns that can suitably be modelled as Behaviour Fragments. These cross-cutting concerns must be able to be well represented by the aspect paradigm and must it must be of benefit to represent them as Behaviour Fragments. In implementing the two MAS scenarios, several agency concern aspects have been implemented. This Section does not contain a comprehensive survey of agency concerns; instead several concerns have been selected in order to demonstrate how they may be represented with aspects.

6.2.1 Communication

One obvious candidate for a behaviour fragment is communication. Specifically, a communication aspect that can be used by Agents to send messages to other Agents. The aspect for such a Communication fragment might use FIPA [1] based messaging or completely different method. Communication is clearly a cross-cutting concern, as the concept of sending messages is common to all agents. Agents on the OPAL platform communicate primarily using JAS (Java Agent Services) based messaging services, which are provided via the Platform. The OPAL platform can also be configured to use other messaging implementations.

The communication aspect implemented can be used to provide messaging functionality via the OPAL platform's FIPA based messaging. The aspect's join points are:

- `public void sendMessage(AgentName name, String message, int id)`

[1] FIPA

The advice for this join point takes the AgentName and message String and constructs a message using the OPAL platform to send the message. The message is sent using the *nzdis − testscenario* ontology, but this aspect might be made more intelligent by assigning different ontologies to what appear to be different types of messages. Alternatively the aspect interface could be extended to incorporate the idea of ontologies. This is not included in the interface in order to keep message sending as generic as possible.

- `public void processRequest(Message message)`

  This method is used for processing incoming messages. The aspect uses advice that is executed around this method and following execution allows the method to proceed. The advice for this pointcut is not really relevant to the functionality of the aspect; the important behaviour related to receiving messages comes via an inter-type declaration. This declaration creates a message filter for that ontology and receiver and applies it to the agent. It also registers the *processRequest* method as the handler for this message filter. This essentially means that any message received by the platform with that agent as the receiver, and using that ontology, invokes the *processRequest* handler for that agent.

- `public void init()`

  The last join point is the *init* method. This is a method incorporated throughout the aspect interfaces that allows the aspects to initiate behaviour upon agent creation. In this case the advice on the *init* join point ensures that the inter-type declaration related to setting up message filters is executed.

6.2.2 Registration

Another behaviour fragment candidate is Registration. Typically Agents register themselves with the directory service local to their platform. A registration behaviour fragment allows agents to augment or replace this behaviour and allows the agent to register with additional or different forms of directory services. Since this behaviour fragment involves augmenting and swapping behaviour dynamically, it is an even better candidate for a behaviour fragment than communication.

The registration aspect is intended to modify or augment the OPAL registration behaviour. OPAL agents are registered with the platform they are created on. This is necessary for the functioning of the agents within the MAS, but the registration aspect can alter the way in which agents register or add to the registration behaviour.

The aspect implemented augments the existing registration process by altering the description of the agent. The change of description details is trivial, but the purpose of implementing this aspect is simply to demonstrate that this is possible, not to provide a complex or real-world example.

The only join point of this aspect is:

```
public void activate()
```

The agent implementing this method it overrides the *activate* method in the *OpalAgent* superclass. The MDA generated method stub first includes a call to *super.activate()* to ensure that the critical registration operations are performed. The advice for the join point executes after this method and can perform any additional registration behaviour; including modifying the registration performed by the previous call to *super.activate()*. The implementation of this aspect does just that by modifying the agent description.

6.2.3 Publication

Another concern involving augmenting or replacing behaviour is the idea of a publication behaviour concern. This fragment could publish the internal information of the Agent in a human readable format. This kind of fragment bears some similarity to logging, which is a classic aspect use case. It benefits a MAS because the behaviour of publication can be abstracted and held separate from the actual Agents themselves.

It is also beneficial as it is platform independent. Some agents publish information via the platforms they occupy, but a publication aspect could independently publish information in a homogeneous way across heterogeneous platforms.

This aspect allows the agent to publish information about its internal state to a publicly available resource. Agents are autonomous entities, and as such generally only access their environment through a limited set of sensors and actuators. This can make determining the state of a MAS, or debugging it, somewhat difficult. The publication aspect provides a window into the agent's internal behaviour, with the information provided to the publication resource available at the agent's discretion.

Two different publication aspects have been implemented; one RSS feed based and one Log4J[2] based. The join point for both aspects is the same:

```
public String publish(String message)
```

This method simply allows the agent to pass in a string which will be published to the resource that any weaved aspects target. The first of the aspects, the *RSSPublicationAspect* generates an RSS feed from the messages passed in. The second, the *Log4JPublicationAspect* passes messages to a defined Log4J instance.

6.3 First MAS Scenario

6.3.1 PIM

Initially a simple and abstract MAS shall be developed. The MAS shall consists of three agents, each with differing agency concerns. The PIM models the MAS using abstract concepts, without describing agents. The entities are named after the properties they contain; they are:

- **ExtrovertTalker** This entity is an extrovert in that it publishes information about its internal state to the world using the *Publication* behaviour fragment; it is also a talker in that it uses the *Communication* behaviour fragment to communicate with other entities.
- **Extrovert** This entity simply publishes information about its state.
- **RegisteredTalker** This entity lets a central registry know about its existence or capabilities via the *Registration* behaviour fragment as well as talking to other entities using the *Communication* behaviour fragment.

Figure 6.1 shows the PIM for this agent scenario.

6.3.2 PSM

After transforming the PIM to the PSM, the model has changed somewhat. The model is now Java and OPAL specific; and contains each entity as Java classes extending from an OpalAgent superclass. This ensures that these classes will now be recognised by the OPAL platform as agents. Each agent also realises the relevant interface for the behaviour fragment associated with it in the PIM. This model uses standard UML notation and could easily be fed into any UML to source code generation tool. Figure 6.2 shows the UML of the PSM.

6.3.3 Resulting Implementation

The final transformation to implementation takes the PSM and creates a Java class for each agent. The Java classes were combined with the interfaces and aspects into an AspectJ project using the Eclipse IDE with the AspectJ development tools. The aspects weaved with the system were the OPAL FIPA based Communication aspect, the RSS feed publication aspect and the additional description Registration aspect. Each aspect transparently provided the agency concern behaviour to the Agents, with the agents oblivious to the aspects themselves.

[2]Log4J - http://logging.apache.org/log4j/1.2/index.html

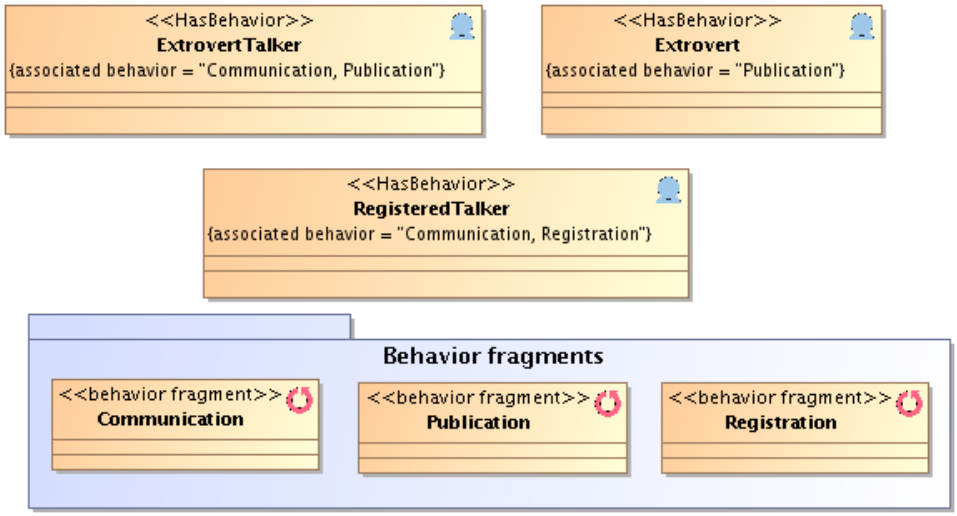

Figure 6.1: The PIM for the first MAS scenario. Contains entities with behaviours and behaviour fragments; associated with tagged values.

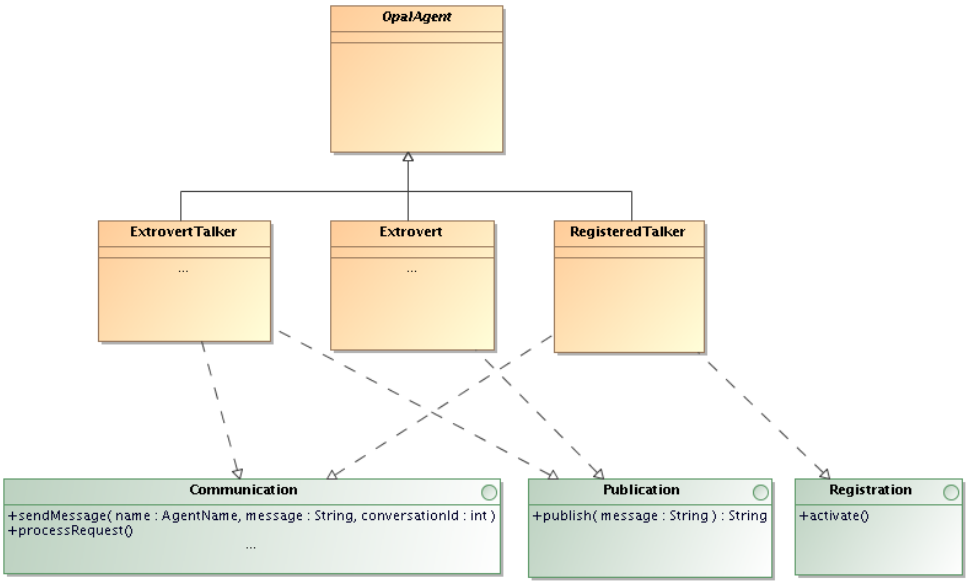

Figure 6.2: The PSM for the first MAS scenario. Contains OPAL agents realising interfaces which provide glue-code for aspects.

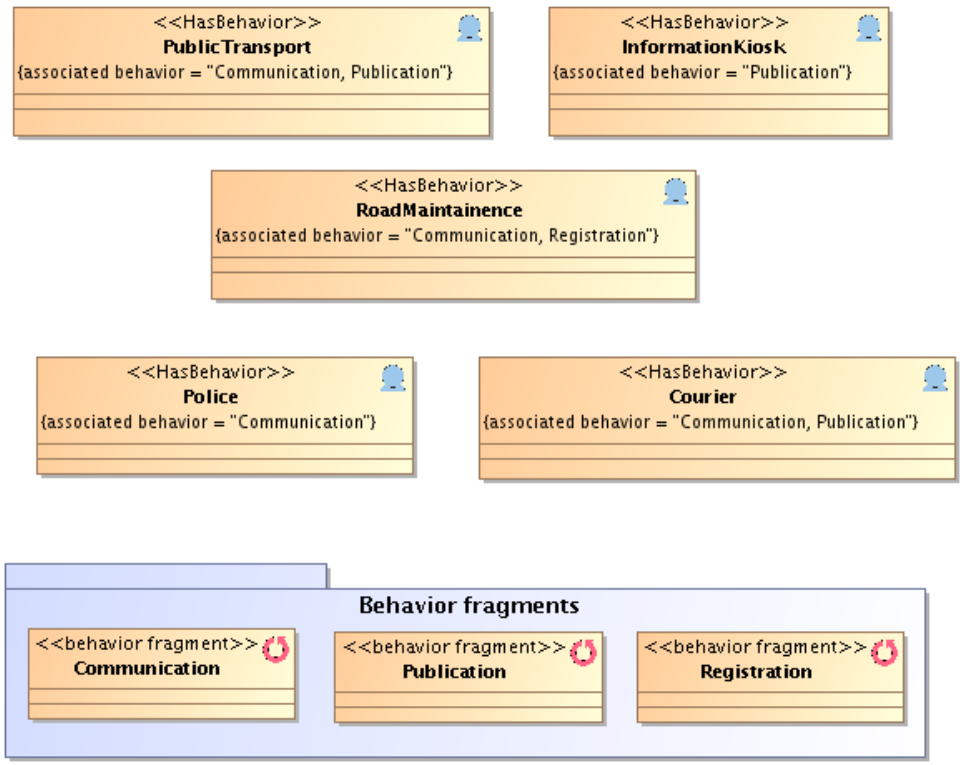

Figure 6.3: The PIM of the second MAS scenario.

6.4 Second MAS Scenario

6.4.1 PIM

The second MAS scenario is less abstract and attempts to model the real world somewhat. The scenario involves a MAS in which there are RoadMaintainence, PublicTransport, InformationKiosk, Police and Courier agents. Each type of agent has a different set of required agency concerns in order to meet its requirements. For instance the PublicTransport agent might need the *Communication* agency concern to communicate with other agents in order to determine if certain routes are closed for maintenance; and it might need the *Publication* concern to provide publically accessible information about its status to commuters. The actual implementation of the agency concerns and the interactions between agent types is not the primary concern here; this scenario is intended to show that a MAS containing many types of agents with different non-functional requirements can be easily modelled and an implementation generated using exactly the same process as the more abstract simple scenario. Figure 6.3 shows the PIM that was modelled.

6.4.2 PSM

The PSM for this scenario follows the same pattern as for the first scenario; and is shown in figure 6.4

6.4.3 Resulting Implementation

The generated implementation was weaved with the same aspects as in the first scenario. The resulting MAS was fully functional when given basic input. This shows that the process proposed is effective in creating different MAS in different scenarios; and that the aspects used to provide agency concerns can be reused across MAS created using the process.

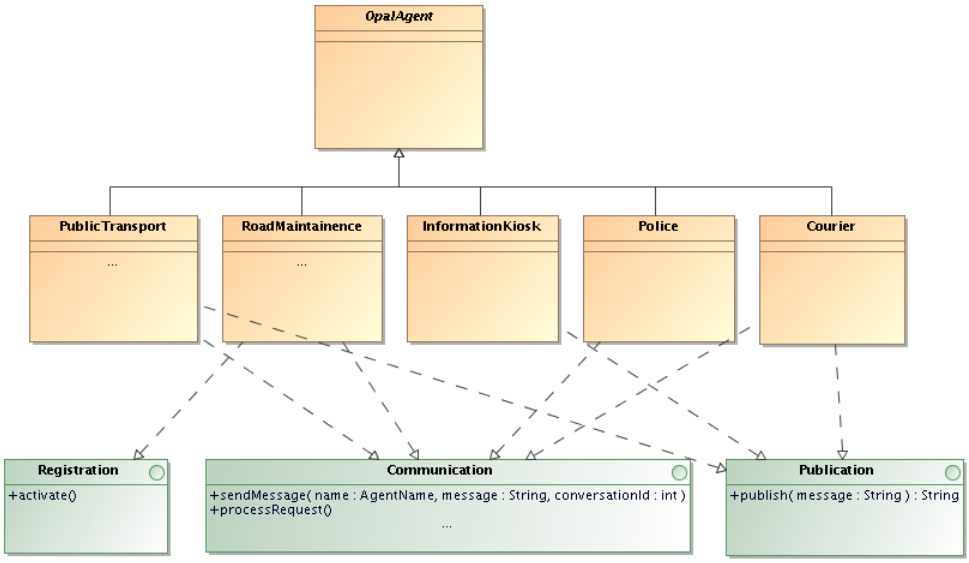

Figure 6.4: The PSM of the second MAS scenario.

6.5 Swapping Aspects

While it is important to demonstrate that the process is interchangeable across MAS, it is also important to demonstrate that the agency concerns aspects can be swapped within the same MAS. This was accomplished using the two implementations of the publication aspect. When applied, both aspects worked well with no modification to the MAS. The aspects were able to be applied individually or applied at the same time, which provided the MAS with both publication implementations.

7 Discussion and Future Work

The results of the evaluation show that the process proposed is suitable for modelling and generating small to medium size MAS. The value of the process lies in the fact that with only the creation of a simple model, a MAS can be generated with agency concerns encapsulated as aspects. The modelling of the MAS is simplistic enough that novice developers could quite easily model a MAS with the need to implement only the functional requirements. Since aspects are used to provide the agency concerns, the developer need only program against the interface of the aspects. This keeps the implementation of the non-functional requirements of the agent separate from the functional requirements and the agent structure. The design and development of the proposed process has resulted in some insight into both the representation of agency concerns as aspects and upon general improvements that could be made to the process. These insights are described within the Sections following.

7.1 Encapsulating agency concerns

The process of developing the small set of agency concerns described in Section 6 has illuminated some of the factors that indicate suitability of an agency concern for aspectualisation. These factors include:

Obliviousness As discussed in Section 3, obliviousness refers to the idea that an agent should be oblivious to the aspect that targets it. Using interfaces for agency concerns facilitates this by ensuring that the agent need only use these interfaces. The aspects use the join points provided by the interfaces and provide the agency concerns transparently. Therefore in order for an agency concern to be a good candidate for representation as an aspect the agent should be oblivious as to which implementation of the aspect is employed.

Obliviousness ensures that aspects providing agency concerns can be swappable. If an agent is oblivious to the aspect then it can more easily be ensured that the glue-code necessary for the aspects is kept clean, simple and to a minimum. The use of interfaces for providing the join points for aspects facilitates obliviousness to such a degree that aspects need not necessarily be used to provide the functionality. Indeed the method stubs realised from the interface could be completed with an implementation or an alternate method for providing the concern could be inserted into the body of the stub, and the agent would be oblivious to the change.

Suitably Abstract Interfaces Having suitably abstract interfaces between aspects and the targeted agents means that the interfaces can support many different types of aspect implementations; including those aspect implementations not yet conceived of. An interface for a concern such as messaging should take arguments that are sufficiently abstract that any implementation of messaging aspect can make use of them.

This was evident to us during the design and implemention of the small set of aspects that were implemented. It is anticipated that with further work a set of guidelines for creating these interfaces could be defined, and the suitability of the interfaces for aspectualisation could perhaps be quantified. Implementing a wider range of aspects could provide insight into this topic, as well as incorporating other MAS platforms into the process. This would reveal the commonalities in the way in which different MAS use display different agency concerns and perhaps reveal a suitable abstraction across MAS.

During development of the aspects this factor was kept in mind, and for the most part this was successful. A possible violation of this factor is the inclusion of the *AgentName* parameter in the communication interface defined. The *AgentName* parameter was included because when sending messages to other agents using OPAL it is necessary. However it is likely that if the communication agency concern was represented by an aspect which implemented SMTP messaging for example, the *AgentName* parameter might no longer be useful for identifying the recipient of a message. Without implementing a range of different communication aspects it is difficult to determine exactly what type of argument would be suitably abstract.

Synchronous vs Asynchronous Whether an operation is synchronous or asynchronous undoubtedly has an impact on its suitability for representation as an aspect. If an operation is synchronous with the core behaviour of an agent then that agent relies on the aspect and hence is no longer oblivious to the aspect. Operations that are asynchronous are not relied on by the agent, and thus the agent is inherently more oblivious to the aspect.

Platform Dependence While only one communication aspect was implemented, the process of implementation revealed an issue related to this concern that potentially has ramifications to other agency concerns. The Communication aspect defined was a FIPA based messaging aspect and it used the functionality provided by the agent's platform in order to provide messaging. Part of the platform's functionality related to a callback type of paradigm, where the agent registered with the platform and the platform would call a certain method within the agent upon receipt of a message for that agent. This worked very well for the platform based messaging, but it seems that different types of messaging could have different paradigms for recieving messages. For instance an aspect that sends messages using email would need to have a mechanism for constantly polling a mail server, as it could no longer simply rely on the platform informing it of new messages.

It seems that part of the issue is that fact that some of these concerns rely heavily on the agent platform, and so incorporating these as aspects in a MAS involves interfering with the workings of the platform. In the case of Communication it could be argued that messaging is the prime component of an Agent platform (the other candidate being some form of directory services). Although it might be useful to provide other messaging services to OPAL agents, it seems that facilitating messaging in such a dynamic and platform independent way lessens the power of the platform and detaches the agents from it somewhat.

Agency Concern Conflicts Due to the limited number of agency concern aspects implemented it was difficult to gain much insight on agency concern conflicts. It is likely however that the implementation of a MAS that used many different agency concern aspects would reveal conflicts between agency concern aspects. Future research could implement such a MAS in order to determine the effect of such conflicting aspects, and to determine ways of avoiding or neutralising these conflicts.

7.2 Implementation to Model Transformation

With any model to implementation process the question of reverse transformations arises. That is, does the model update when changes are made within the implementation? Currently the proposed process does not support this concept, and it would take a considerable amount of effort to incorporate it. The use of a source code generation tool that supported this concept would solve this problem in the case of the implementation to PSM transformation; but the PSM to PIM transformation would require the PIM to be derived from the constructs in the PIM. This could be a formidable task.

The question must be asked however; would this reverse transformation process be desirable? Currently the primary function of the process is to derive a skeleton MAS implementation from a PSM. Since the implementation is generated automatically, it can be ensured that it conforms to specific interfaces so that the aspects can reliably target the MAS. Any changes to the generated implementation could arguably be achieved more simply by modifying the model and regenerating the modified agents within the MAS.

Doing this would require small changes to the model, rather than changes to the implementation that must adhere to the interfaces defined.

Regardless of the motivation for reverse transformation, its implementation would be a considerable task and shall be left for future work.

7.3 Improvements on the Process

The limitations of modelling imposed some limitations on the proposed process that had a small negative effect on its usability. Specifically, the version of MagicDraw available (Personal Edition) did not support the importing of XMI. This leaves a slight gap in the linear process. This was worked around by duplicating the PIM and manually making the changes necessary to give us the PSM expected after the transformation. The XMI was then exported and used as a target for the transformation defined. The result of the transformation was compared with the exported XMI, if they were the same then the transformation was assumed to be successful. Improved tool support could lead to a more streamlined development process.

While aspects have proved highly capable at representing agency concerns it is possible that a more natural way of representing agency concerns exists. It is possible that existing OOP techniques could be more widely applicable. Future research could directly compare the use of aspects and other paradigms in order to conclusively determine if aspects are preferable to other techniques.

Beyond even the issue of the suitability of aspects for representing agency concerns, future work could look at determining whether different architectures are more suitable for representation of agents than the Object Oriented paradigm. It is conceivable that the problem of representing agency concerns separately from the agent themselves might not be an issue if the Object boundary were not part of the problem.

In Section 6 the development of two MAS using proposed process is detailed. These two MAS are both based on the OPAL agent platform using the Java programming language. An important part of the proposed process is the fact that a single PSM can be transformed into multiple PSM, each with a corresponding implementation. Defining MAS on multiple platforms using a single PIM has not been discussed here, but an important next step in development of the process would be to define more transformations from the PIM to different MAS platforms. It is anticipated that implementing further transformations would provide insight as to how agency concerns may be represented on different platforms using aspects, and would make the process more flexible overall. Further development in would also lead to more agency concern aspects being implemented; resulting in a process that could be applicable in more varied MAS scenarios.

7.4 Future Research Direction

To summarise, the proposed approach shows promise but further work is required in order to determine conclusively if it is more valuable than existing techniques. The process needs to be applied to real-world MAS scenarios, further agency concerns aspects need to be developed and different transformations to different platforms need to be incorporated. The broadening of the process, and the insights gained doing so will bring us closer to determining if the proposed process is truely valuable in MAS development.

8 Conclusion

The proposed process leverages the MDA process in the development of a MAS in which Agency concerns are represented with aspects. The process adheres to the MDA process of a series of transformations from a PIM to a PSM and finally into an implementation. The aspects defined to represent agency concerns are reusable between MAS developed using the proposed process. The design of the process, and the example MAS created using it has provided insight as to the suitability of various agency concerns for representation via aspects. Further work is necessary to comprehensively survey agency concerns and quantify suitability for representation via aspects.

The process has proved successful for generation of a MAS on the OPAL agent platform using the Java programming language, and aspects have proven to be competent at representing agency concerns within the generated MAS.

While the proposed process has been successful in the artificial MAS scenarios used to evaluate its effectiveness, the next step should be to apply the process to a variety of real-world scenarios in order to truly determine its value in developing MAS.

Bibliography

[1] The new shorter oxford english dictionary, 1993.

[2] ALDAWUD, O., ELRAD, T., AND BADER, A. Uml profile for aspect-oriented software development, 2003.

[3] ALESSANDRO F. GARCIA, VIVIANE TORRES DA SILVA, C. C. C. J. P. D. L. Engineering multi-agent systems with aspects and patterns. In *Journal of the Brazilian Computer Society* (2002), vol. 8.

[4] ALESSANDRO F. GARCIA, C. J. P. D. L. An aspect-based object-oriented model for multi-agent systems. In *2nd Advanced Separation of Concerns Workshop at ICSE* (2001).

[5] ARAJO, J., AND ANA MOREIRA, ISABEL BRITO, A. R. Aspect-oriented requirements with uml. In *Workshop on "Aspect-oriented Modelling with UML", UML 2002* (2002).

[6] C GONZALEZ, JM MURILLO, P. A. Aspect-oriented analysis: a mda based approach.

[7] FILMAN, R. What is aspect-oriented programming, revisited.

[8] FUENTES, L., PINTO, M., AND VALLECILLO, A. How mda can help designing component- and aspect-based applications.

[9] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science 707* (1993), 406–431.

[10] GARCIA, A., AND DE LUCENA, C. J. P. An aspect-based object-oriented model for multi-agent systems. In *2nd Advanced Separation of Concerns Workshop at ICSE* (2001).

[11] GARCIA, A., KULESZA, U., SANTANNA, C., CHAVEZ, C., AND DE LUCENA, C. J. P. Aspects in agent-oriented software engineering: Lessons learned. In *Proc. 6th Workshop on Agent-Oriented on Software Engineering* (2005).

[12] GRUNDY, J. Multi-perspective specification, design and implementation of software components using aspects, 2000.

[13] HO, W.-M., PENNANEAC'H, F., AND PLOUZEAU, N. Umlaut: A framework for weaving uml-based aspect-oriented designs. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 324.

[14] JACOBSON, I., AND NG, P.-W. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[15] JOAQUIN MILLER, J. M. Mda guide version 1.0.

[16] JOEL CHAMPEAU, FRANCOIS MEKERKE, E. R. Towards a clear definition of patterns, aspects and views in mda. In *EMSISE'03* (ENSIETA - Laboratoire DTN, 29806 Brest Cedex, France, 2003).

[17] JULIO, Y. Y. From goals to aspects: Discovering aspects from requirements goal models.

[18] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. *Lecture Notes in Computer Science 2072* (2001), 327–355.

[19] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[20] KULKARNI, V., AND REDDY, S. Supporting aspects in mda.

[21] MV HECHT, EK PIVETA, M. P. R. P. Aspect-oriented code generation. In *XX Simpsio Brasileiro de Engenharia de Software* (Florianpolis, SC, Brasil, 2006).

[22] PAWLAK, R., SEINTURIER, L., AND RETAILL, J.-P. Foundations of aop for j2ee development.

[23] PURVIS, M., NOWOSTAWSKI, M., AND CRANEFIELD, S. A multi-level approach and infrastructure for agent-oriented software development. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems* (New York, NY, USA, 2002), ACM Press, pp. 88–89.

[24] RASHID, A., MOREIRA, A., AND ARAUJO, J. Modularisation and composition of aspectual requirements, 2003.

[25] ROMAIN ROBBES, NOURY BOURAQADI, S. S. An aspect-based multi-agent system. In *ESUG Conference* (2004).

A Sample Generated Agent

The following is the code generated for a single agent using the process. This the code for the Police agent taken from the scenario defined in Section 6.4.

```
 /*
* Created by OPAL source generation script
* Author: Toby Cox
*/

package agents;

import javax.agent.AgentName;
import nzdis.agent.OpalAgent;
import interfaces.*;
import nzdis.agent.message.Message;
import org.rakiura.micro.SystemAgentLoader;

public class Police extends OpalAgent implements Communication {

    public Police(String displayName){
        super(displayName);
        SystemAgentLoader.newAgent(this);
        this.activate();
        init();
    }

   /**
    * This method is called within the constructor and
    * can be targeted by Aspects wishing to perform some form of init.
    * If new Aspects are applied to a running MAS, it may pay to call this method
    * to ensure the new Aspect functions correctly.
    **/
    public void init(){
    }

    public void sendMessage(AgentName name, String message, int id){
    }

    public void processRequest(Message message){
    }

}
```

B Communication Aspect

The following is the implementation of the Communication aspect as described in Section 6.2.

```
 package aspects;
import javax.agent.AgentName;
import interfaces.*;

import nzdis.agent.OpalAgent;
import nzdis.agent.message.Message;
import nzdis.agent.message.MessageFilter;
import agents.*;

public aspect FIPACommunicationAspect {

public OpalAgent Communication.getAgentInstance(){ return (OpalAgent)this; }
public void Communication.listenForPings(){
Message pattern = new Message(Message.REQUEST);
pattern.set(Message.ONTOLOGY, "nzdis-testscenario");

MessageFilter myBehavior = new MessageFilter(pattern){
public void action(Message aMesg) {
processRequest(aMesg);
}
};

OpalAgent thisAgent = (OpalAgent)this;
thisAgent.getAgent().getGroup().getAgentLoader().newAgent(myBehavior);
}

pointcut sendMessageCut(OpalAgent sender, AgentName agentName, String message, int convo_id):
    call(void Communication.sendMessage(AgentName, String, int))
    && target(sender)
    && args(agentName, message, convo_id);

Object around(OpalAgent sender, AgentName agentName, String message, int convo_id):
sendMessageCut(sender, agentName, message, convo_id) {
System.out.println("Aspect: FIPA Communication Aspect");

Message theMessage = new Message(Message.REQUEST);
theMessage.set(Message.CONTENT, message);
theMessage.set(Message.ONTOLOGY, "nzdis-testscenario");
theMessage.setReceiver(agentName);

theMessage.set(Message.CONVERSATION_ID, convo_id);
```

```
sender.send(theMessage);
return proceed(sender, agentName, message, convo_id);
}

pointcut processRequestCut(OpalAgentof the agent's name reciever, Message message):
    call(void Communication.processRequest(Message))
    && target(reciever)
    && args(message);

Object around(OpalAgent reciever, Message message): processRequestCut(reciever, message) {
System.out.println("Aspect: FIPA Communication Aspect Recieving");

Object content = message.get(Message.CONTENT);
System.out.println(reciever.getDisplayName() + " recieved " + content);


return proceed(reciever, message);
}

// target init method
pointcut initCut(Communication caller):
    call(void Communication.init())
    && target(caller);

Object around(Communication caller): initCut(caller) {
System.out.println("Aspect: FIPA Communication init method");

caller.listenForPings();

return proceed(caller);
}


}
```

PIM to PSM QiQu Script

The following QiQu script is used for the transformation from a PIM to an OPAL specific PSM.

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
<QiQuScript>
<EchoText InfoText="'Performing Agent Transformation'" />

    <LoadDoc FileName="'model/input/HelloAgents.uml'" NewDocRef="$inModel"/>


    <!-- Get the diagram parent element -->
    <SelectFirst NodeRef="$inModel" XPath="//uml:Model[@name='Data']"
SelectedEleRef="$parentNode"/>

<!-- Add the AgentName superclass -->
<CreateEle NodeRef="$parentNode" EleName="'packagedElement'"
NewEleRef="$agentName_class" />
<Set Ref="$agentName_class.type" Value="'uml:Class'"/>
<Set Ref="$agentName_class.xmi:id" Value="'AgentName_class'"/>
<Set Ref="$agentName_class.name" Value="'AgentName'"/>

<!-- Add the OpalAgent class -->
<CreateEle NodeRef="$parentNode" EleName="'packagedElement'"
NewEleRef="$opalAgent" />
<Set Ref="$opalAgent.type" Value="'uml:Class'"/>
<Set Ref="$opalAgent.xmi:id" Value="'OpalAgent_parentClass'"/>
<Set Ref="$opalAgent.name" Value="'OpalAgent'"/>

<!-- Add the Message class -->
<CreateEle NodeRef="$parentNode" EleName="'packagedElement'"
NewEleRef="$message_class" />
<Set Ref="$message_class.type" Value="'uml:Class'"/>
<Set Ref="$message_class.xmi:id" Value="'Message_Class'"/>
<Set Ref="$message_class.name" Value="'Message'"/>

<!-- add all classes as subclasses of OPAL -->
<!-- get an agent stereotype -->
    <For NodeRef="$inModel" XPath="//MASProfile:HasBehavior"
IteratorEleRef="$agentList">
    <Set Ref="key" Value="$agentList.base_Class"/>
<SelectFirst NodeRef="$inModel" XPath="//packagedElement[@xmi:id='#key#']"
SelectedEleRef="$thisAgent"/>
       <EchoText InfoText="'Found ' + $thisAgent.name +
' agent, adding Opal.Agent superclass'"/>
```

```
       <CreateEle NodeRef="$thisAgent" EleName="'generalization'"
NewEleRef="$generalisation" />
       <Set Ref="$generalisation.xmi:id" Value="'generalisation_' + $thisAgent.name"/>
       <Set Ref="$generalisation.isSubstitutable" Value="'false'"/>
       <Set Ref="$generalisation.general" Value="$opalAgent.xmi:id"/>

    </For>

    <!-- create interfaces -->
    <CreateEle NodeRef="$parentNode" EleName="'packagedElement'"
NewEleRef="$publicationInterface"/>
    <Set Ref="$publicationInterface.xmi:type" Value="'uml:Interface'"/>
    <Set Ref="$publicationInterface.xmi:id" Value="'publicationInterface'"/>
    <Set Ref="$publicationInterface.name" Value="'Publication'"/>

<!-- Create publication method -->
<CreateEle NodeRef="$publicationInterface"
EleName="'ownedOperation'" NewEleRef="$operation" />
<Set Ref="$operation.xmi:id" Value="$publicationInterface.name + '_publicationOperation'"/>
<Set Ref="$operation.visibility" Value="'public'"/>
<Set Ref="$operation.name" Value="'publish'"/>
<EchoText InfoText="'created publication operation'"/>

<CreateEle NodeRef="$operation" EleName="'ownedParameter'"
NewEleRef="$parameter" />
<Set Ref="$parameter.xmi:id" Value="$publicationInterface.name + '_returnParameter'"/>
<Set Ref="$parameter.visibility" Value="'public'"/>
<Set Ref="$parameter.direction" Value="'return'"/>

<CreateEle NodeRef="$parameter" EleName="'type'" NewEleRef="$type" />
<Set Ref="$type.xmi:type" Value="'uml:PrimitiveType'"/>
<Set Ref="$type.href" Value="'pathmap://UML_LIBRARIES/
UMLPrimitiveTypes.library.uml#String'"/>

<CreateEle NodeRef="$operation" EleName="'ownedParameter'" NewEleRef="$parameter2" />
<Set Ref="$parameter2.xmi:id" Value="$publicationInterface.name + '_parameter'"/>
<Set Ref="$parameter2.name" Value="'message'"/>
<Set Ref="$parameter2.visibility" Value="'public'"/>
<Set Ref="$parameter2.direction" Value="'inout'"/>

<CreateEle NodeRef="$parameter2" EleName="'type'" NewEleRef="$type" />
<Set Ref="$type.xmi:type" Value="'uml:PrimitiveType'"/>
<Set Ref="$type.href" Value="'pathmap://UML_LIBRARIES/
UMLPrimitiveTypes.library.uml#String'"/>

<!-- create communication interface -->
<CreateEle NodeRef="$parentNode" EleName="'packagedElement'"
NewEleRef="$communicationInterface"/>
<Set Ref="$communicationInterface.xmi:type" Value="'uml:Interface'"/>
    <Set Ref="$communicationInterface.xmi:id" Value="'communicationInterface'"/>
    <Set Ref="$communicationInterface.name" Value="'Communication'"/>
<!-- Create sendMessage method -->
<CreateEle NodeRef="$communicationInterface" EleName="'ownedOperation'"
```

```
NewEleRef="$operation" />
<Set Ref="$operation.xmi:id" Value="$communicationInterface.name
+ '_sendMessageOperation'"/>
<Set Ref="$operation.visibility" Value="'public'"/>
<Set Ref="$operation.name" Value="'sendMessage'"/>

<!-- parameters -->
<!-- agent name -->
<CreateEle NodeRef="$operation" EleName="'ownedParameter'"
NewEleRef="$parameter2" />
<Set Ref="$parameter2.xmi:id" Value="$operation.name
+ '_agentName_parameter'"/>
<Set Ref="$parameter2.name" Value="'name'"/>
<Set Ref="$parameter2.visibility" Value="'public'"/>
<Set Ref="$parameter2.direction" Value="'inout'"/>
<Set Ref="$parameter2.type" Value="$agentName_class.xmi:id"/>

<!-- message -->
<CreateEle NodeRef="$operation" EleName="'ownedParameter'"
NewEleRef="$parameter2" />
<Set Ref="$parameter2.xmi:id" Value="$operation.name + '_message_parameter'"/>
<Set Ref="$parameter2.name" Value="'message'"/>
<Set Ref="$parameter2.visibility" Value="'public'"/>
<Set Ref="$parameter2.direction" Value="'inout'"/>

<CreateEle NodeRef="$parameter2" EleName="'type'" NewEleRef="$type" />
<Set Ref="$type.xmi:type" Value="'uml:PrimitiveType'"/>
<Set Ref="$type.href" Value="'pathmap://UML_LIBRARIES/
UMLPrimitiveTypes.library.uml#String'"/>

<!-- convo id -->
<CreateEle NodeRef="$operation" EleName="'ownedParameter'"
NewEleRef="$parameter2" />
<Set Ref="$parameter2.xmi:id" Value="$operation.name
+ '_convo_id_parameter'"/>
<Set Ref="$parameter2.name" Value="'id'"/>
<Set Ref="$parameter2.visibility" Value="'public'"/>
<Set Ref="$parameter2.direction" Value="'inout'"/>

<CreateEle NodeRef="$parameter2" EleName="'type'" NewEleRef="$type" />
<Set Ref="$type.xmi:type" Value="'uml:PrimitiveType'"/>
<Set Ref="$type.href" Value="'pathmap://UML_LIBRARIES/
UMLPrimitiveTypes.library.uml#int'"/>

<EchoText InfoText="'created sendMessage operation'"/>

<!-- Create processRequest method -->
<CreateEle NodeRef="$communicationInterface"
EleName="'ownedOperation'" NewEleRef="$operation" />
<Set Ref="$operation.xmi:id" Value="$communicationInterface.name
+ '_processRequestOperation'"/>
<Set Ref="$operation.visibility" Value="'public'"/>
<Set Ref="$operation.name" Value="'processRequest'"/>
<EchoText InfoText="'created processRequest operation'"/>
```

```
<!-- Message parameter -->
<CreateEle NodeRef="$operation" EleName="'ownedParameter'"
NewEleRef="$parameter2" />
<Set Ref="$parameter2.xmi:id" Value="$operation.name
+ '_message_parameter'"/>
<Set Ref="$parameter2.name" Value="'message'"/>
<Set Ref="$parameter2.visibility" Value="'public'"/>
<Set Ref="$parameter2.direction" Value="'inout'"/>
<Set Ref="$parameter2.type" Value="$message_class.xmi:id"/>

    <!-- create registration interface -->
<CreateEle NodeRef="$parentNode" EleName="'packagedElement'"
NewEleRef="$registrationInterface"/>
<Set Ref="$registrationInterface.xmi:type" Value="'uml:Interface'"/>
    <Set Ref="$registrationInterface.xmi:id" Value="'registrationInterface'"/>
    <Set Ref="$registrationInterface.name" Value="'Registration'"/>
<!-- Create activate method -->
<CreateEle NodeRef="$registrationInterface" EleName="'ownedOperation'"
NewEleRef="$operation" />
<Set Ref="$operation.xmi:id" Value="$registrationInterface.name
+ '_activateOperation'"/>
<Set Ref="$operation.visibility" Value="'public'"/>
<Set Ref="$operation.name" Value="'activate'"/>


<!-- Publication -->
<For NodeRef="$inModel" XPath="//MASProfile:HasBehavior[contains
(@associatedbehavior, 'Publication')]" IteratorEleRef="$relatedToBehavior">
<SelectFirst NodeRef="$parentNode"
XPath="//packagedElement[@xmi:id='#$relatedToBehavior.base_Class#']"
SelectedEleRef="$associatedAgent" />
<!-- realise interface -->
<CreateEle NodeRef="$associatedAgent" EleName="'interfaceRealization'"
NewEleRef="$realization"/>
<Set Ref="$realization.xmi:id" Value="$associatedAgent.name
+ '_publication_realization'"/>
<Set Ref="$realization.name" Value="''"/>
<Set Ref="$realization.supplier" Value="$publicationInterface.xmi:id"/>
<Set Ref="$realization.client" Value="$associatedAgent.id"/>
<Set Ref="$realization.contract" Value="$publicationInterface.xmi:id"/>

</For>

<!-- Communication -->
<For NodeRef="$inModel" XPath="//MASProfile:HasBehavior[contains
(@associatedbehavior, 'Communication')]" IteratorEleRef="$relatedToBehavior">
<SelectFirst NodeRef="$parentNode"
XPath="//packagedElement[@xmi:id='#$relatedToBehavior.base_Class#']"
SelectedEleRef="$associatedAgent" />
<!-- realise interface -->
<CreateEle NodeRef="$associatedAgent" EleName="'interfaceRealization'"
NewEleRef="$realization"/>
<Set Ref="$realization.xmi:id" Value="$associatedAgent.name
```

```
+ '_communication_realization'"/>
<Set Ref="$realization.name" Value="''"/>
<Set Ref="$realization.supplier" Value="$communicationInterface.xmi:id"/>
<Set Ref="$realization.client" Value="$associatedAgent.id"/>
<Set Ref="$realization.contract" Value="$communicationInterface.xmi:id"/>
</For>

<!-- Registration -->
<For NodeRef="$inModel" XPath="//MASProfile:HasBehavior[contains(
@associatedbehavior, 'Registration')]" IteratorEleRef="$relatedToBehavior">
<SelectFirst NodeRef="$parentNode"
XPath="//packagedElement[@xmi:id='#$relatedToBehavior.base_Class#']"
SelectedEleRef="$associatedAgent" />
<!-- realise interface -->
<CreateEle NodeRef="$associatedAgent" EleName="'interfaceRealization'"
NewEleRef="$realization"/>
<Set Ref="$realization.xmi:id" Value="$associatedAgent.name + '_registration_realization'"/>
<Set Ref="$realization.name" Value="''"/>
<Set Ref="$realization.supplier" Value="$registrationInterface.xmi:id"/>
<Set Ref="$realization.client" Value="$associatedAgent.id"/>
<Set Ref="$realization.contract" Value="$registrationInterface.xmi:id"/>
    <EchoText InfoText="'Realised reg interface'"/>
</For>

    <!-- delete behavior fragments -->
    <For NodeRef="$inModel" XPath="//MASProfile:behaviorfragment"
IteratorEleRef="$thisBehaviorFragment">
     <SelectFirst NodeRef="$thisBehaviorFragment"
XPath="//packagedElement[@xmi:id='#$thisBehaviorFragment.base_Class#']"
SelectedEleRef="$toDelete"/>
     <Delete Ref="$toDelete"/>
    </For>

<!-- Clean up stereotypes etc -->
<For NodeRef="$inModel" XPath="//MASProfile:*" IteratorEleRef="$profiles">
<Delete Ref="$profiles"/>
</For>




    <SaveDoc FileName="'model/output/HelloAgentsPSM.xml'"
DocRef="$inModel" Encoding="'ISO-8859-1'"/>
</QiQuScript>
```

D PSM to Implementation QiQu Script

The following QiQu script is used to generate an Java implementation of an OPAL PSM.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<QiQuScript>
  <LoadDoc FileName="'model/output/HelloAgentsPSM.xml'" NewDocRef="$inModel"/>
<For NodeRef="$inModel" XPath="//packagedElement[@xmi:type='uml:Class']"
  IteratorEleRef="$theclass">
  <EchoText InfoText="'Found: ' + $theclass.name"/>
  <!-- creating the file -->
<CreateFile NewFileRef="$lfile"/>
<!-- creating the class header -->
<PrintToFile FileRef="$lfile" Value="'/*'"/>
<PrintToFile FileRef="$lfile" Value="'* Created by OPAL source generation script'"/>
<PrintToFile FileRef="$lfile" Value="'* Author: Toby Cox'"/>
<PrintToFile FileRef="$lfile" Value="'*/'"/>
<PrintToFile FileRef="$lfile" Value="''"/>
<PrintToFile FileRef="$lfile" Value="'package agents;'"/>
<PrintToFile FileRef="$lfile" Value="''"/>
<PrintToFile FileRef="$lfile" Value="'import javax.agent.AgentName;'"/>
<PrintToFile FileRef="$lfile" Value="'import nzdis.agent.OpalAgent;'"/>
<PrintToFile FileRef="$lfile" Value="'import interfaces.*;'"/>
<PrintToFile FileRef="$lfile" Value="'import nzdis.agent.message.Message;'"/>
<PrintToFile FileRef="$lfile" Value="'import org.rakiura.micro.SystemAgentLoader;'"/>

<PrintToFile FileRef="$lfile" Value="''"/>

  <!-- get generalisation if available-->
  <SelectFirst NodeRef="$theclass" XPath="generalization" ]
SelectedEleRef="$generalisation"/>
  <PrintToFile FileRef="$lfile" Value="'public class '
+ $theclass.name" Crlf="'false'"/>
  <If Condition="not(equals($generalisation,'null'))">
   <EchoText InfoText="'Found generalisation to: '
+ $generalisation.general"/>
   <SelectFirst NodeRef="$inModel"
XPath="//packagedElement[@xmi:id='#$generalisation.general#']"
SelectedEleRef="$parent"/>
   <PrintToFile FileRef="$lfile" Value="' extends '
+ $parent.name" Crlf="'false'"/>
  </If>
<!-- add interfaces -->
```

```
<SelectFirst NodeRef="$inModel" XPath="//uml:Model[@name='Data']"
SelectedEleRef="$parentNode"/>
<CreateEle NodeRef="$parentNode" EleName="'temp'" NewEleRef="$tempParamList"/>
<Set Ref="$tempParamList.contents" Value="' implements '" />
<For NodeRef="$theclass" XPath="interfaceRealization"
IteratorEleRef="$thisRealization">
<SelectFirst NodeRef="$inModel" XPath="
//packagedElement[@xmi:id='#$thisRealization.supplier#']"
SelectedEleRef="$interface"/>
<Set Ref="$tempParamList.contents" Value="$tempParamList.contents
+ $interface.name + ', '" />
</For>
<Set Ref="$tempParamList.contents" Value="$tempParamList.contents + 'end'"/>
<PrintToFile FileRef="$lfile" Value="replace($tempParamList.contents, ', end', '')" Crlf="'false'"/>

<PrintToFile FileRef="$lfile" Value="' {'"/>
<PrintToFile FileRef="$lfile" Value="''"/>

<!-- constructor -->
<PrintToFile FileRef="$lfile" Value="'    public ' + $theclass.name
+ '(String displayName){'"/>
<PrintToFile FileRef="$lfile" Value="'        super(displayName);'"/>
<PrintToFile FileRef="$lfile" Value="'        SystemAgentLoader.newAgent(this);'"/>
<!-- if registration -->
<For NodeRef="$theclass" XPath="interfaceRealization"
IteratorEleRef="$thisRealization">
<SelectFirst NodeRef="$inModel" XPath="//packagedElement[@xmi:id='#$thisRealization.supplier#']"
SelectedEleRef="$interface"/>
<If Condition="equals($interface.name, 'Registration')">
<PrintToFile FileRef="$lfile" Value="'        super.activate();'"/>
</If>
</For>
<PrintToFile FileRef="$lfile" Value="'        this.activate();'"/>
<PrintToFile FileRef="$lfile" Value="'        init();'"/>
<PrintToFile FileRef="$lfile" Value="'    }'"/>
<PrintToFile FileRef="$lfile" Value="''"/>

<!-- init method -->
<PrintToFile FileRef="$lfile" Value="'   /**'"/>
<PrintToFile FileRef="$lfile" Value="'    * This method is called within the constructor and '"/>
<PrintToFile FileRef="$lfile" Value="'    * can be targeted by Aspects wishing to perform some form
of init.'"/>
<PrintToFile FileRef="$lfile" Value="'    * If new Aspects are applied to a running MAS, it may
pay to call this method'"/>
<PrintToFile FileRef="$lfile" Value="'    * to ensure the new Aspect functions correctly.'"/>
<PrintToFile FileRef="$lfile" Value="'    **/'"/>
<PrintToFile FileRef="$lfile" Value="'    public void init(){'"/>
<PrintToFile FileRef="$lfile" Value="'    }'"/>
<PrintToFile FileRef="$lfile" Value="''"/>

<!-- get the classes related behaviors and make the methods related -->
<For NodeRef="$theclass" XPath="interfaceRealization"
IteratorEleRef="$thisRealization">
<SelectFirst NodeRef="$inModel"
```

```
XPath="//packagedElement[@xmi:id='#$thisRealization.supplier#']"
SelectedEleRef="$interface"/>

<!-- create methods -->
<For NodeRef="$interface" XPath="ownedOperation" IteratorEleRef="$thisOp">
<PrintToFile FileRef="$lfile" Value="'    '
+ $thisOp.visibility + ' '" Crlf="'false'"/>
<!-- get return value -->
<SelectFirst NodeRef="$thisOp" XPath="ownedParameter[@direction='return']"
SelectedEleRef="$returnParam"/>
<If Condition="not(equals($returnParam, 'null'))">
<SelectFirst NodeRef="$returnParam"
XPath="type" SelectedEleRef="$type"/>
<PrintToFile FileRef="$lfile" Value="replace($type.href,'pathmap.*#', '') + ' '"
 Crlf="'false'" />
</If> <!-- else -->
<If Condition="equals($returnParam, 'null')">
<PrintToFile FileRef="$lfile" Value="'void '" Crlf="'false'" />
</If>

<PrintToFile FileRef="$lfile" Value="$thisOp.name + '('" Crlf="'false'" />


<!-- get parameters -->
<!-- create temp element --> <!-- Get the diagram parent element -->
<Set Ref="$tempParamList.contents" Value="''" />

<For NodeRef="$thisOp" XPath="ownedParameter[@direction='inout']"
 IteratorEleRef="$thisParam">
<SelectFirst NodeRef="$thisParam" XPath="type"
SelectedEleRef="$type"/>
<!-- if a custom class type -->
<If Condition="not(equals($thisParam.type, 'null'))">
<SelectFirst NodeRef="$inModel"
 XPath="//packagedElement[@xmi:id='#$thisParam.type#']"
 SelectedEleRef="$paramType"/>

<Set Ref="$tempParamList.contents"
Value="$tempParamList.contents + $paramType.name"/>
</If>
<!-- if not a custom class type -->
<If Condition="equals($thisParam.type, 'null')">

<Set Ref="$tempParamList.contents"
Value="$tempParamList.contents
+ replace($type.href,'pathmap.*#', '')"/>
</If>

<Set Ref="$tempParamList.contents" Value="$tempParamList.contents + ' '+
 $thisParam.name + ', '"/>
</For>

<Set Ref="$tempParamList.contents" Value="$tempParamList.contents
+ 'end){'"/>
```

```
<PrintToFile FileRef="$lfile" Value="replace(replace(
$tempParamList.contents, ', end', ''), 'end', '')" />
<!-- if that old return parameter up there was not void,
then at least return null -->
<If Condition="not(equals($returnParam, 'null'))">
<PrintToFile FileRef="$lfile" Value="'        return null;'" />
</If>
<PrintToFile FileRef="$lfile" Value="'    }'" />
<PrintToFile FileRef="$lfile" Value="''" />
</For>
</For>
<PrintToFile FileRef="$lfile" Value="'}'" />



  <SaveFile FileName="'model/output/src/' + $theclass.name + '.java'" FileRef="$lfile"/>
</For>
</QiQuScript>
```